`rfuA` Signed 4-byte integer, big-endian byte ordering.
Reserved for future use. Set to zero.

`data` Compressed CDF data.

Field	Size	Comments
RecordSize	4 bytes	
RecordType	4 bytes	
CPRoffset	4 bytes	
uSize	4 bytes	
rfuA	4 bytes	
data	variable	Size is RecordSize - 20 bytes.

Figure 2.11: Compressed CDF Record (CCR)

2.10 Compressed Parameters Record

A Compressed Paramters Record (CPR) is used to keep the information as the compression method and level used to create a CDF or variable. This record is pointed to by either a CCR or a VDR. When a compression is applied to the whole CDF, the CPR is pointed to by the CCR. If a compression is only applied to a variable, a CPR is pointed to by a VDR. Currently, only Run-Length Encoding (RLE), Huffman (HUFF), Adaptive Huffman (AHUFF) and GNU GZIP compression algorithms are supported.[14]

Each CPR, as shown in Figure 2.12, contains the following contiguous fields...

`RecordSize` Signed 4-byte integer, big-endian byte ordering.
The size in bytes of this CPR (including this field).

`RecordType` Signed 4-byte integer, big-endian byte ordering.
The value `11` which identifies this as a CPR.

`cType` Signed 4-byte integer, big-endian byte ordering.
Type of compression.

`rfuA` Signed 4-byte integer, big-endian byte ordering.
Reserved for future use. Set to zero.

`pCount` Signed 4-byte integer, big-endian byte ordering.
Compression parameter count. Currently, it is `1`.

`cParms` Signed 4-byte integer, big-endian byte ordering.
Compression level. For RLE, HUFF and AHUFF, cParms[0] is 0. For GZIP, it is between 1 and 9.

[14] Due to a huge memory requirement, the GZIP compression is disabled for the PCs running the 16-bit DOS/Windows 3.x.

Field	Size	Comments
`RecordSize`	4 bytes	
`RecordType`	4 bytes	
`cType`	4 bytes	
`rfuA`	4 bytes	
`pCount`	4 bytes	
`cParms`	variable	Size depends on `pCount`

Figure 2.12: Compressed Paramters Record (CPR)

2.11 Sparseness Paramters Record

A Sparseness parameters Record (SPR) is used to store sparse array information used by a variable record in a CDF. Currently, it has not yet been implemented in the V2.6 distribtuion.

Each SPR, as shown in Figure 2.13, contains the following contiguous fields...

`RecordSize` Signed 4-byte integer, big-endian byte ordering.
The size in bytes of this SPR (including this field).

`RecordType` Signed 4-byte integer, big-endian byte ordering.
The value `11` which identifies this as a SPR.

`sArraysType` Signed 4-byte integer, big-endian byte ordering.
include the magic numbers.

`rfuA` Signed 4-byte integer, big-endian byte ordering.
Reserved for future use. Set to zero.

`pCount` Signed 4-byte integer, big-endian byte ordering.
Sparseness parameter count.

`sArraysParms` Signed 4-byte integer, big-endian byte ordering.
Parameters for sparseness arrays.

Field	Size	Comments
`RecordSize`	4 bytes	
`RecordType`	4 bytes	
`sArraysType`	4 bytes	
`rfuA`	4 bytes	
`pCount`	4 bytes	
`sArraysParms`	variable	Size depends on `pCount`.

Figure 2.13: Sparseness Parameters Record (SPR)

2.12 Compressed Variable Values Record

A Compressed Variable Values Record (CVVR) is used to store one section of compressed variable values records (VVRs) for a variable in a single-file CDF. This section of VVRs while uncompressed are contiguous in the physical file or scratch temporary file. CVVRs will not exist in multi-file CDFs.

Each CVVR, as shown in Figure 2.14, contains the following contiguous fields...

`RecordSize` Signed 4-byte integer, big-endian byte ordering.
The size in bytes of this CVVR (including this field).

`RecordType` Signed 4-byte integer, big-endian byte ordering.
The value `12` which identifies this as a CVVR.

`rfuA` Signed 4-byte integer, big-endian byte ordering.
Reserved for future use. Set to zero.

`cSize` Signed 4-byte integer, big-endian byte ordering.
Size in bytes of the compressed data which follows.

`data` Compressed data.

Field	Size	Comments
`RecordSize`	4 bytes	
`RecordType`	4 bytes	
`rfuA`	4 bytes	
`cSize`	4 bytes	
`data`	variable	Size is specified in `cSize`

Figure 2.14: Compressed Variable Values Record (CVVR)

2.13 Unused Internal Record

Internal records in the dotCDF file of a CDF may become unused due to a number of reasons. When that occurs, the internal record is marked as being unused and is placed on a double-linked list of Unused Internal Records (UIRs). The `UIRhead` field of the GDR contains the file offset of the first UIR. The first UIR contains the file offset of the next UIR and so on. The last UIR contains a file offset of `0x00000000` as the file offset of the next UIR (to indicate the end of the UIRs). Likewise, the last UIR contains the file offset of the previous UIR and so on. The first UIR contains a file offset of `0x00000000` as the file offset of the previous UIR (to indicate the start of the UIRs).

Each UIR, as shown in Figure 2.15, contains the following contiguous fields...

`RecordSize` Signed 4-byte integer, big-endian byte ordering.
The size in bytes of this UIR (including this field).

`RecordType`	Signed 4-byte integer, big-endian byte ordering. The value `-1` which identifies this as a UIR. (See the section on UUIRs below for a slight complication.)
`NextUIR`	Signed 4-byte integer, big-endian byte ordering. The file offset of the next UIR. The last UIR will contain a file offset of `0x00000000` in this field (to indicate the end of the UIRs).
`PrevUIR`	Signed 4-byte integer, big-endian byte ordering. The file offset of the previous UIR. The first UIR will contain a file offset of `0x00000000` in this field (to indicate the start of the UIRs).
`Remainder`	Zero or more unused bytes which constitute the remainder of the UIR. The contents of this field are undefined.

Field	**Size**	**Comments**
`RecordSize`	4 bytes	
`RecordType`	4 bytes	
`NextUIR`	4 bytes	
`PrevUIR`	4 bytes	
`Remainder`	variable	Size depends on the size of this UIR.

Figure 2.15: Unused Internal Record (UIR)

It is possible to have internal records in the dotCDF file of a CDF that are unused but are not considered UIRs. Let's call them Unsociable Unused Internal Records (UUIRs) because they are not on the double-linked list of UIRs that begins at the file offset contained in the `UIRhead` field of the GDR. CDFs prior to CDF V2.5 will contain only UUIRs because UIRs were not yet supported. Beginning with CDF V2.5 UUIRs may also exist due to special circumstances (e.g, if an internal record that is no longer needed is less than 16 bytes which means that it is too small to be made a UIR).

Each UUIR, as shown in Figure 2.16, contains the following contiguous fields…

`RecordSize`	Signed 4-byte integer, big-endian byte ordering. The size in bytes of this UUIR (including this field).
`RecordType`	Signed 4-byte integer, big-endian byte ordering. The value `-1` which identifies this as a UUIR. Unfortunately this is the same value as that used for UIRs. UUIRs are distinguished from UIRs by the fact that they are not on the double-linked list of UIRs.
`Remainder`	Zero or more unused bytes which constitute the remainder of the UUIR. The contents of this field are undefined.

Field	Size	Comments
`RecordSize`	4 bytes	
`RecordType`	4 bytes	
`Remainder`	variable	Size depends on the size of this UUIR.

Figure 2.16: Unsociable Unused Internal Record (UUIR)

Chapter 3

Variable Files

In multi-file CDFs, variable records are stored in separate files — one per variable. Assuming a base name of <cdfname>, the CDF would consist of the file named <cdfname>.cdf,[1] a file named <cdfname>.v<i> for each rVariable (where <i> is the rVariable number), and a file named <cdfname>.z<j> for each zVariable (where <j> is the zVariable number). Note that variables are numbered beginning with zero (0). For example, a multi-file CDF named sample having three rVariables would consist of the files sample.cdf, sample.v0, sample.v1, and sample.v2.

Within each variable file are stored the corresponding variable records. The variable records are stored contiguously beginning with record number zero (0) with no gaps in the record numbering. The number of records will correspond to the MaxRec field of the variable's VDR (described in Section 2.6). The size of each variable record will be the same and depends on the dimensionality, dimension variances, data type, and number of elements per value of the corresponding variable. These properties are discussed in Chapter 4. The encoding of the values in each variable record depends on the encoding of the CDF (which is stored in the Encoding field of the CDR). The possible encodings are described in Chapter 5.

[1]On VMS and DOS systems, the file names/extensions would be uppercase.

Chapter 4

Variable Records

Variable records contain the values written to a variable. Each variable record contains one variable array. The physical layout of a variable array depends on the dimensionality and dimension variances of the variable and the variable majority of the CDF. The dimensionality of an rVariable is contained in the `rNumDims` and `rDimSizes` fields of the GDR. The dimensionality of a zVariable is contained in the `zNumDims` and `rDimSizes` fields of the corresponding zVDR. Dimension variances are contained in the `DimVarys` field of the corresponding rVDR/zVDR. The CDF's variable majority is contained in bit 0 of the `Flags` field of the CDR. Note also that each variable array value consists of some number of elements of the variable's data type. A variable's data type and number of elements of that data type at each variable value are contained in the `DataType` and `NumElems` fields of the corresponding rVDR/zVDR.

Dimension variances allow a conceptual view of a physical variable array. For each array dimension, if the corresponding dimension variance is `TRUE`, then the dimension actually exists. If the dimension variance is `FALSE`, then the dimension is virtual and is not physically stored. This would probably be a good time for an example. Assume a variable with the following characteristics...

Data Type	CDF_REAL4
Number of Elements	1
Number of Dimensions	2
Dimension Sizes	3,5
Dimension Variances	TRUE,FALSE

The conceptual view of this variable array is that of a `3` by `5` 2-dimensional array (represented by the syntax `2:[3,5]`). The `TRUE`,`FALSE` dimension variances indicate that the first dimension is real (physically stored) but that the second dimension is virtual (not physically stored). When an application accesses a value in this variable array two dimension indices are specified, one per dimension (represented by the syntax `(i,j)` where `i` and `j` are the dimension indices). The first index is used to physically position to a value in the array (because the corresponding dimension variance is `TRUE`). The second index, however, is essentially ignored because the corresponding dimension variance of `FALSE` indicates that the second dimension is virtual and is not physically stored. Conceptually, all values along the second dimension are the same (and are the one value which is physically stored). This means that `(i,0)`, `(i,1)`, `(i,2)`, `(i,3)`, and `(i,4)` all map to the same physical location in the variable array for any given first dimension index (`i`). For this variable record stored at a file offset of `n` (in the dotCDF file or a variable file), the conceptual values would map to the physical values as follows...

File Offset of Physical Value	Indices of Conceptual Value(s)
n	(0,0),(0,1),(0,2),(0,3),(0,4)
n+4	(1,0),(1,1),(1,2),(1,3),(1,4)
n+8	(2,0),(2,1),(2,2),(2,3),(2,4)

Note that only three values are physically stored with each consisting of four bytes (which is the size of one element of the `CDF_REAL4` data type).

Had the dimension variances been `FALSE`,`TRUE` instead, the conceptual to physical mapping would be as follows...

File Offset of Physical Value	Indices of Conceptual Value(s)
n	(0,0),(1,0),(2,0)
n+4	(0,1),(1,1),(2,1)
n+8	(0,2),(1,2),(2,2)
n+12	(0,3),(1,3),(2,3)
n+16	(0,4),(1,4),(2,4)

In this case five values are physically stored and it is along the first dimension that all values are conceptually the same.

It is not until two or more of the dimensions are physically stored (having dimension variances of `TRUE`) that the variable majority of the CDF has an effect. Row majority means that the first dimension changes slowest in the physical storage of the array and column majority means that the last dimension changes the slowest. Assume that in our example the dimension variances are `TRUE`,`TRUE`. The physical layout of the array values for each variable majority would be as follows...

File Offset of Physical Value	Indices of Conceptual Value(s), Row Majority	Indices of Conceptual Value(s), Column Majority
n	(0,0)	(0,0)
n+4	(0,1)	(1,0)
n+8	(0,2)	(2,0)
n+12	(0,3)	(0,1)
n+16	(0,4)	(1,1)
n+20	(1,0)	(2,1)
n+24	(1,1)	(0,2)
n+28	(1,2)	(1,2)
n+32	(1,3)	(2,2)
n+36	(1,4)	(0,3)
n+40	(2,0)	(1,3)
n+44	(2,1)	(2,3)
n+48	(2,2)	(0,4)
n+52	(2,3)	(1,4)
n+56	(2,4)	(2,4)

Note that an application's conceptual view of the variable array does not depend on the variable majority. When an application accesses the value at indices `(i,j)` the proper value will be accessed. The physical location of that value, however, depends very much on the variable majority of the CDF.

0-dimensional and 1-dimensional variables are relatively simple. The variable array of a 0-dimesional variable consists of one physically stored value. 1-dimensional variable arrays are stored as a vector of one or more physical values when the dimension variance is `TRUE` or just a single physically stored value when the dimension variance is FALSE (with all of the values along the dimension being conceptually the same).

When a variable value consists of more than one element (e.g., character data having the CDF_CHAR data type), all of the elements of that value are stored contiguously with the first element being at the lowest file offset.

The size in bytes of a variable record is the product of the size in bytes of the data type, the number of elements of the data type at each variable value, and the size of each dimension having a variance of TRUE.

As a final example consider a variable with the following characteristics...

Data Type	CDF_CHAR
Number of Elements	5
Number of Dimensions	3
Dimension Sizes	2,3,4
Dimension Variances	TRUE,FALSE,TRUE

The conceptual value to physical value mapping for each majority would be as follows...

File Offset of Physical Value	Indices of Conceptual Value(s), Row Majority	Indices of Conceptual Value(s), Column Majority
n	(0,0,0),(0,1,0),(0,2,0)	(0,0,0),(0,1,0),(0,2,0)
n+5	(0,0,1),(0,1,1),(0,2,1)	(1,0,0),(1,1,0),(1,2,0)
n+10	(0,0,2),(0,1,2),(0,2,2)	(0,0,1),(0,1,1),(0,2,1)
n+15	(0,0,3),(0,1,3),(0,2,3)	(1,0,1),(1,1,1),(1,2,1)
n+20	(1,0,0),(1,1,0),(1,2,0)	(0,0,2),(0,1,2),(0,2,2)
n+25	(1,0,1),(1,1,1),(1,2,1)	(1,0,2),(1,1,2),(1,2,2)
n+30	(1,0,2),(1,1,2),(1,2,2)	(0,0,3),(0,1,3),(0,2,3)
n+35	(1,0,3),(1,1,3),(1,2,3)	(1,0,3),(1,1,3),(1,2,3)

In this example each variable record would consist of 40 bytes (which is the product of the size in bytes of one element of the data type [1], the number of elements of the data type at each variable value [5], the size of the first dimension [2], and the size of the last dimension [4]).

Chapter 5

Encodings

5.1 Data Representations

5.1.1 Bits

The following sections will refer to fields of one or more bits. In all cases the lowest numbered bit is the least significant.

5.1.2 Bytes

A byte consists of eight bits numbered 0 through 7 (with bit 0 being the least significant). When values consisting of more than one byte are referenced, the lowest numbered byte is stored at the lowest file offset. (The lowest numbered byte is not necessarily the least significant byte.)

5.1.3 Integers

Integers consist of one, two, or four bytes. 1-byte integers contain eight bits numbered 0 through 7. 2-byte integers contain 16 bits numbered 0 through 15. 4-byte integers contain 32 bits numbered 0 through 31. In each case bit 0 is the least significant bit.

Signed integers are stored in two's-complement binary notation. For 1-byte integers this provides a range of values from `-128` through `127`. For 2-byte integers this provides a range of values from `-32768` through `32767`. For 4-byte integers this provides a range of values from `-2147483648` through `2147483647`.

Unsigned integers are stored in binary notation. For 1-byte integers this provides a range of values from `0` through `255`. For 2-byte integers this provides a range of values from `0` through `65535`. For 4-byte integers this provides a range of values from `0` through `4294967295`.

Little-endian integers are stored with the least-significant byte first (i.e., at the lowest file offset) and big-endian integers are stored with the most-significant byte first. Table 5.1 illustrates little-endian and big-endian byte orderings.

	Little-Endian		Big-Endian	
	Byte/Offset	Contents	Byte/Offset	Contents
2-byte integer	0	bits 0-7	0	bits 8-15
	1	bits 8-15	1	bits 0-7
4-byte integer	0	bits 0-7	0	bits 24-31
	1	bits 8-15	1	bits 16-23
	2	bits 16-23	2	bits 8-15
	3	bits 24-31	3	bits 0-7

Table 5.1: Little-Endian vs. Big-Endian

5.1.4 Floating-Point

Several floating-point encodings are possible in a CDF. Each is described in the following sections. Note that a loss of precision may occur when converting between the various encodings because of differences in the number of mantissa bits. Likewise, there are differences in the minimum and maximum magnitudes which may be represented because of differences in the number of exponent bits. Appendix A illustrates how the different single-precision floating-point encodings map to actual floating-point values and Appendix B illustrates the same for double-precision floating-point encodings.

IEEE 754 Single-Precision Floating-Point

IEEE[1] 754 single-precision floating-point values consist of four bytes containing one sign bit, eight exponent bits (numbered 0 through 7), and 23 mantissa bits (numbered 0 through 22). IEEE 754 single-precision floating-point values are stored in one of two ways: little-endian or big-endian. The arrangements of the bits are shown in Tables 5.2 and 5.3, respectively.

Byte/Offset	Bit(s)	Contents
0	0-7	mantissa bits 0-7
1	0-7	mantissa bits 8-15
2	0-6	mantissa bits 16-22
	7	exponent bit 0
3	0-6	exponent bits 1-7
	7	sign bit (negative if set)

Table 5.2: IEEE 754, Single-Precision Floating-Point, Little-Endian

Digital's `F_FLOAT` Single-Precision Floating-Point

Digital's[2] `F_FLOAT` single-precision floating-point values consist of four bytes containing one sign bit, eight exponent bits (numbered 0 through 7), and 23 mantissa bits (numbered 0 through 22). The arrangement of the bits is shown in Table 5.4.

[1]The Institute of Electrical and Electronics Engineers, Inc.
[2]Digital Equipment Corporation

Byte/Offset	Bit(s)	Contents
0	0-6	exponent bits 1-7
	7	sign bit (negative if set)
1	0-6	mantissa bits 16-22
	7	exponent bit 0
2	0-7	mantissa bits 8-15
3	0-7	mantissa bits 0-7

Table 5.3: IEEE 754, Single-Precision Floating-Point, Big-Endian

Byte/Offset	Bit(s)	Contents
0	0-6	mantissa bits 16-22
	7	exponent bit 0
1	0-6	exponent bits 1-7
	7	sign bit (negative if set)
2	0-7	mantissa bits 0-7
3	0-7	mantissa bits 8-15

Table 5.4: Digital's `F_FLOAT`, Single-Precision Floating-Point

IEEE 754 Double-Precision Floating-Point

IEEE 754 double-precision floating-point values consist of eight bytes containing one sign bit, eleven exponent bits (numbered 0 through 10), and 52 mantissa bits (numbered 0 through 51). IEEE 754 double-precision floating-point values are stored in one of two ways: little-endian or big-endian. The arrangements of the bits are shown in Tables 5.5 and 5.6, respectively.

Byte/Offset	Bit(s)	Contents
0	0-7	mantissa bits 0-7
1	0-7	mantissa bits 8-15
2	0-7	mantissa bits 16-23
3	0-7	mantissa bits 24-31
4	0-7	mantissa bits 32-39
5	0-7	mantissa bits 40-47
6	0-3	mantissa bits 48-51
	4-7	exponent bits 0-3
7	0-6	exponent bits 4-10
	7	sign bit (negative if set)

Table 5.5: IEEE 754, Double-Precision Floating-Point, Little-Endian

Digital's `D_FLOAT` Double-Precision Floating-Point

Digital's `D_FLOAT` double-precision floating-point values consist of eight bytes containing one sign bit, eight exponent bits (numbered 0 through 7), and 55 mantissa bits (numbered 0 through 54). The arrangement of the bits is shown in Table 5.7.

Byte/Offset	Bit(s)	Contents
0	0-6	exponent bits 4-10
	7	sign bit (negative if set)
1	0-3	mantissa bits 48-51
	4-7	exponent bits 0-3
2	0-7	mantissa bits 40-47
3	0-7	mantissa bits 32-39
4	0-7	mantissa bits 24-31
5	0-7	mantissa bits 16-23
6	0-7	mantissa bits 8-15
7	0-7	mantissa bits 0-7

Table 5.6: IEEE 754, Double-Precision Floating-Point, Big-Endian

Byte/Offset	Bit(s)	Contents
0	0-6	mantissa bits 48-54
	7	exponent bit 0
1	0-6	exponent bits 1-7
	7	sign bit (negative if set)
2	0-7	mantissa bits 32-39
3	0-7	mantissa bits 40-47
4	0-7	mantissa bits 16-23
5	0-7	mantissa bits 24-31
6	0-7	mantissa bits 0-7
7	0-7	mantissa bits 8-15

Table 5.7: Digital's `D_FLOAT`, Double-Precision Floating-Point

Digital's `G_FLOAT` Double-Precision Floating-Point

Digital's `G_FLOAT` double-precision floating-point values consist of eight bytes containing one sign bit, eleven exponent bits (numbered 0 through 10), and 52 mantissa bits (numbered 0 through 51). The arrangement of the bits is shown in Table 5.8.

Byte/Offset	Bit(s)	Contents
0	0-3	mantissa bits 48-51
	4-7	exponent bits 0-3
1	0-6	exponent bits 4-10
	7	sign bit (negative if set)
2	0-7	mantissa bits 32-39
3	0-7	mantissa bits 40-47
4	0-7	mantissa bits 16-23
5	0-7	mantissa bits 24-31
6	0-7	mantissa bits 0-7
7	0-7	mantissa bits 8-15

Table 5.8: Digital's `G_FLOAT`, Double-Precision Floating-Point

5.2 Control Information

Two types of data are stored in a CDF — control information and application data. Control information is used to manage the application data stored in a CDF. A user application generally does not have access to the control information.[3] Throughout this document, individual pieces of control information will also be referred to as "internal values."

5.2.1 Integer Values

Integer control information is stored in 4-byte signed or unsigned integers with big-endian byte ordering. Two's-complement is used for signed integers.

5.2.2 Character Strings

Character string control information is stored using the ASCII character set. The character strings are `NUL`-terminated[4] unless the number of characters is exactly equal to the size of the field containing the character string.

5.3 Application Data

Application data consists of attribute entry values (commonly referred to as "metadata") and variable values (simply referred to as "data"). Note that some of the control information stored in a CDF could also be considered application metadata (e.g., attribute and variable names, the CDF's data encoding and variable majority, and variable dimensionalities). For the purpose of this document, however, these internal values will be considered control information.

Application data values are stored according to the data encoding of the CDF. A CDF's data encoding is stored in the CDF Descriptor Record (CDR) described in Section 2.2. Application data values are also stored as one of the supported CDF data types. Table 5.9 lists the supported data types and the corresponding internal values used to identify each data type.

The possible data encodings for a CDF correspond to the platforms on which the CDF software distribution is supported. Table 5.10 lists the currently supported data encodings along with the corresponding internal values used to identify each data encoding.

Table 5.11 shows how each of the supported data types are stored for a particular data encoding. Note that many of the data encodings are actually stored in the same way. Table 5.11 shows the equivalent data encodings.

[3] An exception to this would be the indexing statistics provided to an application by the CDF library for variables in a single-file CDF.

[4] The ASCII `NUL` character (an integer value of `0x00`).

Data Type	Internal Value	Description
CDF_INT1	1	1-byte, signed integer.
CDF_INT2	2	2-byte, signed integer.
CDF_INT4	4	4-byte, signed integer.
CDF_UINT1	11	1-byte, unsigned integer.
CDF_UINT2	12	2-byte, unsigned integer.
CDF_UINT4	14	4-byte, unsigned integer.
CDF_BYTE[a]	41	1-byte, signed integer.
CDF_REAL4	21	4-byte, single-precision floating-point.
CDF_REAL8	22	8-byte, double-precision floating-point.
CDF_FLOAT[b]	44	4-byte, single-precision floating-point.
CDF_DOUBLE[c]	45	8-byte, double-precision floating-point.
CDF_EPOCH[d]	31	8-byte, double-precision floating-point.
CDF_CHAR	51	1-byte, signed character (ASCII).[e]
CDF_UCHAR	52	1-byte, unsigned character (ASCII).[e]

[a] CDF_BYTE values are equivalent to CDF_INT1 values.
[b] CDF_FLOAT values are equivalent to CDF_REAL4 values.
[c] CDF_DOUBLE values are equivalent to CDF_REAL8 values.
[d] CDF_EPOCH values are equivalent to CDF_REAL8 values. CDF_EPOCH is used to store date/time values (as the number of milliseconds since 1-Jan-0000 00:00:00.000).
[e] Both signed and unsigned character data types are provided for applications that may want to distinguish between the two. Note that attribute entries and variable values of this type are never NUL-terminated.

Table 5.9: Supported Data Types

Data Encoding	Internal Value	Description
NETWORK_ENCODING	1	XDR[a] representation.
SUN_ENCODING	2	Sun representation.
VAX_ENCODING	3	VAX representation.
DECSTATION_ENCODING	4	DECstation representation.
SGi_ENCODING	5	SGi representation.
IBMPC_ENCODING	6	IBM PC representation.
IBMRS_ENCODING	7	IBM RS-6000 representation.
MAC_ENCODING	9	Macintosh representation.
HP_ENCODING	11	HP 9000 series representation.
NeXT_ENCODING	12	NeXT representation.
ALPHAOSF1_ENCODING	13	DEC Alpha/OSF1 representation.
ALPHAVMSd_ENCODING	14	DEC Alpha/OpenVMS representation. Double-precision floating-point values in D_FLOAT encoding.
ALPHAVMSg_ENCODING	15	DEC Alpha/OpenVMS representation. Double-precision floating-point values in G_FLOAT encoding.
ALPHAVMSi_ENCODING	16	DEC Alpha/OpenVMS representation. Single/Double-precision floating-point values in IEEE 754 encoding.

[a] eXternal Data Representation.

Table 5.10: Supported Data Encodings

Data Encoding(s)[a]	Data Type(s) CDF_BYTE CDF_INT1 CDF_UINT1	CDF_INT2 CDF_UINT2	CDF_INT4 CDF_UINT4	CDF_REAL4 CDF_FLOAT	CDF_REAL8 CDF_DOUBLE CDF_EPOCH	CDF_CHAR CDF_UCHAR
NETWORK SUN HP NeXT MAC SGi IBMRS	1-byte integer	2-byte integer, big-endian	4-byte integer, big-endian	IEEE 754 single-precision floating-point, big-endian	IEEE 754 double-precision floating-point, big-endian	ASCII character set
DECSTATION IBMPC ALPHAOSF1 ALPHAVMSi	1-byte integer	2-byte integer, little-endian	4-byte integer, little-endian	IEEE 754 single-precision floating-point, little-endian	IEEE 754 double-precision floating-point, little-endian	ASCII character set
VAX ALPHAVMSd	1-byte integer	2-byte integer, little-endian	4-byte integer, little-endian	Digital's F_FLOAT single-precision floating-point	Digital's D_FLOAT double-precision floating-point	ASCII character set
ALPHAVMSg	1-byte integer	2-byte integer, little-endian	4-byte integer, little-endian	Digital's F_FLOAT single-precision floating-point	Digital's G_FLOAT double-precision floating-point	ASCII character set

[a] _ENCODING is implied at the end of each encoding.

Table 5.11: Data Encodings vs. Data Types

Appendix A

Single-Precision Floating-Point

This appendix presents the exponent and mantissa values for a variety of single-precision floating-point values using Digital's F_FLOAT and the IEEE 754 encoding. The sign bit is not shown but when the sign bit is clear (0x0) the floating-point value is positive and when the sign bit is set (0x1) the value is negative. Section 5.1.4 illustrates how these exponent and mantissa values are arranged in a particular single-precision floating-point value.

	Digital's F_FLOAT		IEEE 754	
Value	**Exp.**	**Mantissa**	**Exp.**	**Mantissa**
0.0000000000e+00	0x00	0x000000[1]	0x00	0x000000
0.0000000000e+00	0x00	0x000001[1]		
0.0000000000e+00	0x00	0x000002[1]		
. . .				
0.0000000000e+00	0x00	0x7FFFFE[1]		
0.0000000000e+00	0x00	0x7FFFFF[1]		
1.4012984643e-45			0x00	0x000001
2.8025969286e-45			0x00	0x000002
4.2038953930e-45			0x00	0x000003
5.6051938573e-45			0x00	0x000004
. . .				
2.9387302719e-39			0x00	0x1FFFFC
2.9387316732e-39			0x00	0x1FFFFD
2.9387330745e-39			0x00	0x1FFFFE
2.9387344758e-39			0x00	0x1FFFFF

2.9387358771e-39	0x01	0x000000	0x00	0x200000
2.9387362274e-39	0x01	0x000001		
2.9387365777e-39	0x01	0x000002		
2.9387369280e-39	0x01	0x000003		
2.9387372784e-39	0x01	0x000004	0x00	0x200001
2.9387376287e-39	0x01	0x000005		
2.9387379790e-39	0x01	0x000006		
2.9387383293e-39	0x01	0x000007		
2.9387386797e-39	0x01	0x000008	0x00	0x200002
.				
.				
.				
5.8774689515e-39	0x01	0x7FFFF8	0x00	0x3FFFFE
5.8774693018e-39	0x01	0x7FFFF9		
5.8774696522e-39	0x01	0x7FFFFA		
5.8774700025e-39	0x01	0x7FFFFB		
5.8774703528e-39	0x01	0x7FFFFC	0x00	0x3FFFFF
5.8774707031e-39	0x01	0x7FFFFD		
5.8774710535e-39	0x01	0x7FFFFE		
5.8774714038e-39	0x01	0x7FFFFF		
5.8774717541e-39	0x02	0x000000	0x00	0x400000
5.8774724548e-39	0x02	0x000001		
5.8774731554e-39	0x02	0x000002	0x00	0x400001
5.8774738561e-39	0x02	0x000003		
5.8774745567e-39	0x02	0x000004	0x00	0x400002
.				
.				
.				
1.1754939304e-38	0x02	0x7FFFFA	0x00	0x7FFFFD
1.1754940005e-38	0x02	0x7FFFFB		
1.1754940706e-38	0x02	0x7FFFFC	0x00	0x7FFFFE
1.1754941406e-38	0x02	0x7FFFFD		
1.1754942107e-38	0x02	0x7FFFFE	0x00	0x7FFFFF
1.1754942808e-38	0x02	0x7FFFFF		
1.1754943508e-38	0x03	0x000000	0x01	0x000000
1.1754944910e-38	0x03	0x000001	0x01	0x000001
1.1754946311e-38	0x03	0x000002	0x01	0x000002
1.1754947712e-38	0x03	0x000003	0x01	0x000003
.				
.				
.				
1.7014114290e+38	0xFF	0x7FFFFC	0xFD	0x7FFFFC
1.7014115304e+38	0xFF	0x7FFFFD	0xFD	0x7FFFFD
1.7014116318e+38	0xFF	0x7FFFFE	0xFD	0x7FFFFE
1.7014117332e+38	0xFF	0x7FFFFF	0xFD	0x7FFFFF

1.7014118346e+38	0xFE	0x000000
1.7014120374e+38	0xFE	0x000001
1.7014122403e+38	0xFE	0x000002
1.7014124431e+38	0xFE	0x000003
.		
.		
.		
3.4028228579e+38	0xFE	0x7FFFFC
3.4028230607e+38	0xFE	0x7FFFFD
3.4028232636e+38	0xFE	0x7FFFFE
3.4028234664e+38	0xFE	0x7FFFFF
Infinity	0xFF	0x000000[2]
NaN	0xFF	0x000001[3]
NaN	0xFF	0x000002[3]
.		
.		
.		
NaN	0xFF	0x7FFFFE[3]
NaN	0xFF	0x7FFFFF[3]

Note that not all single-precision floating-point values can be represented in both encodings. Several ranges of floating-point values, as well as some individual values, are of interest...

`0.0000000000e+00`

When an `F_FLOAT` value has an exponent of `0x00`, the floating-point value represented is `0.0000000000e+00` regardless of the value of the mantissa.

`1.4012984643e-45` through `2.9387344758e-39`

These values can only be represented with the IEEE 754 encoding. Their magnitudes are too small for the `F_FLOAT` encoding.

`2.9387358771e-39` through `5.8774714038e-39`

The `F_FLOAT` encoding has more precision in this range. Four times as many `F_FLOAT` values fall into this range as do IEEE 754 values.

`5.8774717541e-39` through `1.1754942808e-38`

The `F_FLOAT` encoding also has more precision in this range. Twice as many `F_FLOAT` values fall into this range as do IEEE 754 values.

`1.1754943508e-38` through `1.7014117332e+38`

The `F_FLOAT` and IEEE 754 encodings have equal precision through this range.

[1] If the sign bit is set (`-0.0`), a `%SYSTEM-F-ROPRAND` fatal error (on VAXes running VMS/OpenVMS) or a `%SYSTEM-F-HPARITH` fatal error (on DEC Alphas running OpenVMS) will occur if the value is used.

[2] `-Infinity` if the sign bit is set.

[3] `-NaN` if the sign bit is set.

`1.7014118346e+38` through `3.4028234664e+38`

These values can only be represented with the IEEE 754 encoding. Their magnitudes are too large for the `F_FLOAT` encoding.

`Infinity`

This value exists only in the IEEE 754 encoding.

`NaN`

Not a Number. These non-values exist only in the IEEE 754 encoding.

Appendix B

Double-Precision Floating-Point

This appendix presents the exponent and mantissa values for a variety of double-precision floating-point values using Digital's G_FLOAT, Digital's D_FLOAT, and the IEEE 754 encoding. The sign bit is not shown but when the sign bit is clear (0x0) the floating-point value is positive and when the sign bit is set (0x1) the value is negative. Section 5.1.4 illustrates how these exponent and mantissa values are arranged in a particular double-precision floating-point value.

	Digital's G_FLOAT		Digital's D_FLOAT		IEEE 754	
Value	Exp.	Mantissa	Exp.	Mantissa	Exp.	Mantissa
0.00000000000000000e+000	0x000	0x0000000000000[1]	0x000	0x00000000000000[1]	0x000	0x0000000000000
0.00000000000000000e+000	0x000	0x0000000000001[1]	0x000	0x00000000000001[1,2]		
0.00000000000000000e+000	0x000	0x0000000000002[1]	0x000	0x00000000000002[1,2]		
.						
.						
.						
0.00000000000000000e+000	0x000	0xFFFFFFFFFFFFE[1]	0x000	0x7FFFFFFFFFFFFE[1,2]		
0.00000000000000000e+000	0x000	0xFFFFFFFFFFFFF[1]	0x000	0x7FFFFFFFFFFFFF[1,2]		
4.94065645841246544e-324					0x000	0x0000000000001
9.88131291682493088e-324					0x000	0x0000000000002
.						
.						
.						
5.56268464626799358e-309					0x000	0x3FFFFFFFFFFFE
5.56268464626799852e-309					0x000	0x3FFFFFFFFFFFF
5.56268464626800346e-309	0x001	0x0000000000000			0x000	0x4000000000000
5.56268464626800469e-309	0x001	0x0000000000001				
5.56268464626800593e-309	0x001	0x0000000000002				
5.56268464626800716e-309	0x001	0x0000000000003				

5.56268464626800840e-309	0x001	0x0000000000004			0x000	0x4000000000001
5.56268464626800963e-309	0x001	0x0000000000005				
5.56268464626801087e-309	0x001	0x0000000000006				
5.56268464626801210e-309	0x001	0x0000000000007				
5.56268464626801334e-309	0x001	0x0000000000008			0x000	0x4000000000002
.						
.						
.						
1.11253692925359970e-308	0x001	0xFFFFFFFFFFFF8			0x000	0x7FFFFFFFFFFFE
1.11253692925359983e-308	0x001	0xFFFFFFFFFFFF9				
1.11253692925359995e-308	0x001	0xFFFFFFFFFFFFA				
1.11253692925360007e-308	0x001	0xFFFFFFFFFFFFB				
1.11253692925360020e-308	0x001	0xFFFFFFFFFFFFC			0x000	0x7FFFFFFFFFFFF
1.11253692925360032e-308	0x001	0xFFFFFFFFFFFFD				
1.11253692925360044e-308	0x001	0xFFFFFFFFFFFFE				
1.11253692925360057e-308	0x001	0xFFFFFFFFFFFFF				
1.11253692925360069e-308	0x002	0x0000000000000			0x000	0x8000000000000
1.11253692925360094e-308	0x002	0x0000000000001				
1.11253692925360119e-308	0x002	0x0000000000002			0x000	0x8000000000001
1.11253692925360143e-308	0x002	0x0000000000003				
1.11253692925360168e-308	0x002	0x0000000000004			0x000	0x8000000000002
.						
.						
.						
2.22507385850719990e-308	0x002	0xFFFFFFFFFFFFA			0x000	0xFFFFFFFFFFFFD
2.22507385850720015e-308	0x002	0xFFFFFFFFFFFFB				
2.22507385850720039e-308	0x002	0xFFFFFFFFFFFFC			0x000	0xFFFFFFFFFFFFE
2.22507385850720064e-308	0x002	0xFFFFFFFFFFFFD				
2.22507385850720089e-308	0x002	0xFFFFFFFFFFFFE			0x000	0xFFFFFFFFFFFFF
2.22507385850720114e-308	0x002	0xFFFFFFFFFFFFF				
2.22507385850720138e-308	0x003	0x0000000000000			0x001	0x0000000000000
2.22507385850720188e-308	0x003	0x0000000000001			0x001	0x0000000000001
.						
.						
.						
2.93873587705571812e-039	0x380	0xFFFFFFFFFFFFE			0x37E	0xFFFFFFFFFFFFE
2.93873587705571844e-039	0x380	0xFFFFFFFFFFFFF			0x37E	0xFFFFFFFFFFFFF
2.93873587705571877e-039	0x381	0x0000000000000	0x001	0x00000000000000	0x37F	0x0000000000000
2.93873587705571885e-039			0x001	0x00000000000001		
2.93873587705571893e-039			0x001	0x00000000000002		
2.93873587705571901e-039			0x001	0x00000000000003		
2.93873587705571910e-039			0x001	0x00000000000004		
2.93873587705571918e-039			0x001	0x00000000000005		
2.93873587705571926e-039			0x001	0x00000000000006		
2.93873587705571934e-039			0x001	0x00000000000007		

2.93873587705571942e-039	0x381	0x0000000000001	0x001	0x00000000000008	0x37F	0x0000000000001
2.93873587705571950e-039			0x001	0x00000000000009		
2.93873587705571959e-039			0x001	0x0000000000000A		
2.93873587705571967e-039			0x001	0x0000000000000B		
2.93873587705571975e-039			0x001	0x0000000000000C		
2.93873587705571983e-039			0x001	0x0000000000000D		
2.93873587705571991e-039			0x001	0x0000000000000E		
2.93873587705571999e-039			0x001	0x0000000000000F		
2.93873587705572007e-039	0x381	0x0000000000002	0x001	0x00000000000010	0x37F	0x0000000000002
2.93873587705572016e-039			0x001	0x00000000000011		
2.93873587705572024e-039			0x001	0x00000000000012		
2.93873587705572032e-039			0x001	0x00000000000013		
2.93873587705572040e-039			0x001	0x00000000000014		

.
.
.

5.87747175411143623e-039	0x381	0xFFFFFFFFFFFFE	0x001	0x7FFFFFFFFFFFF0	0x37F	0xFFFFFFFFFFFFE
5.87747175411143632e-039			0x001	0x7FFFFFFFFFFFF1		
5.87747175411143640e-039			0x001	0x7FFFFFFFFFFFF2		
5.87747175411143648e-039			0x001	0x7FFFFFFFFFFFF3		
5.87747175411143656e-039			0x001	0x7FFFFFFFFFFFF4		
5.87747175411143664e-039			0x001	0x7FFFFFFFFFFFF5		
5.87747175411143672e-039			0x001	0x7FFFFFFFFFFFF6		
5.87747175411143681e-039			0x001	0x7FFFFFFFFFFFF7		
5.87747175411143689e-039	0x381	0xFFFFFFFFFFFFF	0x001	0x7FFFFFFFFFFFF8	0x37F	0xFFFFFFFFFFFFF
5.87747175411143697e-039			0x001	0x7FFFFFFFFFFFF9		
5.87747175411143705e-039			0x001	0x7FFFFFFFFFFFFA		
5.87747175411143713e-039			0x001	0x7FFFFFFFFFFFFB		
5.87747175411143721e-039			0x001	0x7FFFFFFFFFFFFC		
5.87747175411143730e-039			0x001	0x7FFFFFFFFFFFFD		
5.87747175411143738e-039			0x001	0x7FFFFFFFFFFFFE		
5.87747175411143746e-039			0x001	0x7FFFFFFFFFFFFF		

.
.
.

1.70141183460469182e+038			0x0FF	0x7FFFFFFFFFFFEB		
1.70141183460469185e+038			0x0FF	0x7FFFFFFFFFFFEC		
1.70141183460469187e+038			0x0FF	0x7FFFFFFFFFFFED		
1.70141183460469189e+038			0x0FF	0x7FFFFFFFFFFFEE		
1.70141183460469192e+038			0x0FF	0x7FFFFFFFFFFFEF		
1.70141183460469194e+038	0x47F	0xFFFFFFFFFFFFE	0x0FF	0x7FFFFFFFFFFFF0	0x47D	0xFFFFFFFFFFFFE
1.70141183460469196e+038			0x0FF	0x7FFFFFFFFFFFF1		
1.70141183460469199e+038			0x0FF	0x7FFFFFFFFFFFF2		
1.70141183460469201e+038			0x0FF	0x7FFFFFFFFFFFF3		
1.70141183460469203e+038			0x0FF	0x7FFFFFFFFFFFF4		
1.70141183460469206e+038			0x0FF	0x7FFFFFFFFFFFF5		
1.70141183460469208e+038			0x0FF	0x7FFFFFFFFFFFF6		
1.70141183460469210e+038			0x0FF	0x7FFFFFFFFFFFF7		
1.70141183460469213e+038	0x47F	0xFFFFFFFFFFFFF	0x0FF	0x7FFFFFFFFFFFF8	0x47D	0xFFFFFFFFFFFFF

1.70141183460469215e+038		0x0FF 0x7FFFFFFFFFFF9	
1.70141183460469218e+038		0x0FF 0x7FFFFFFFFFFFA	
1.70141183460469220e+038		0x0FF 0x7FFFFFFFFFFFB	
1.70141183460469222e+038		0x0FF 0x7FFFFFFFFFFFC[3]	
1.70141183460469225e+038		0x0FF 0x7FFFFFFFFFFFD[3]	
1.70141183460469227e+038		0x0FF 0x7FFFFFFFFFFFE[3]	
1.70141183460469229e+038		0x0FF 0x7FFFFFFFFFFFF[3]	
1.70141183460469232e+038	0x480 0x0000000000000		0x47E 0x0000000000000
1.70141183460469270e+038	0x480 0x0000000000001		0x47E 0x0000000000001
.			
.			
.			
8.98846567431157754e+307	0x7FF 0xFFFFFFFFFFFFE		0x7FD 0xFFFFFFFFFFFFE
8.98846567431157854e+307	0x7FF 0xFFFFFFFFFFFFF		0x7FD 0xFFFFFFFFFFFFF
8.98846567431157954e+307			0x7FE 0x0000000000000
8.98846567431158153e+307			0x7FE 0x0000000000001
.			
.			
.			
1.79769313486231551e+308			0xF7E 0xFFFFFFFFFFFFE
1.79769313486231571e+308			0xF7E 0xFFFFFFFFFFFFF
Infinity			0x7FF 0x0000000000000[4]
NaN			0x7FF 0x0000000000001[5]
NaN			0x7FF 0x0000000000002[5]
.			
.			
.			
NaN			0x7FF 0xFFFFFFFFFFFFE[5]
NaN			0x7FF 0xFFFFFFFFFFFFF[5]

Note that not all double-precision floating-point values can be represented in all encodings. Several ranges of floating-point values, as well as some individual values, are of interest...

`0.0000000000e+00`

When a `G_FLOAT` or `D_FLOAT` value has an exponent of `0x00`, the floating-point value represented is `0.00000000000000000e+00` regardless of the value of the mantissa.

[1] If the sign bit is set (`-0.0`), a `%SYSTEM-F-ROPRAND` fatal error (on VAXes running VMS/OpenVMS) or a `%SYSTEM-F-HPARITH` fatal error (on DEC Alphas running OpenVMS) will occur if the value is used.

[2] Even if the sign bit is clear, a `%SYSTEM-F-HPARITH` fatal error will occur if the value is used on a DEC Alpha running OpenVMS.

[3] If the sign bit is set or clear, a `%SYSTEM-F-HPARITH` fatal error will occur if the value is used on a DEC Alpha running OpenVMS.

[4] `-Infinity` if the sign bit is set.

[5] `-NaN` if the sign bit is set.

`4.94065645841246544e-324` through `5.56268464626799852e-309`

These values can only be represented with the IEEE 754 encoding. Their magnitudes are too small for the `G_FLOAT` and `D_FLOAT` encodings.

`5.56268464626800346e-309` through `1.11253692925360057e-308`

These values can only be represented with the `G_FLOAT` and IEEE 754 encodings. Their magnitudes are too small for the `D_FLOAT` encoding. In this range the `G_FLOAT` encoding has more precision than the IEEE 754 encoding. Four times as many `G_FLOAT` values fall into this range as do IEEE 754 values.

`1.11253692925360069e-308` through `2.22507385850720114e-308`

These values can only be represented with the `G_FLOAT` and IEEE 754 encodings. Their magnitudes are too small for the `D_FLOAT` encoding. In this range the `G_FLOAT` encoding has more precision than the IEEE 754 encoding. Twice as many `G_FLOAT` values fall into this range as do IEEE 754 values.

`2.22507385850720138e-308` through `2.93873587705571844e-039`

These values can only be represented with the `G_FLOAT` and IEEE 754 encodings. Their magnitudes are too small for the `D_FLOAT` encoding. In this range the `G_FLOAT` and IEEE 754 encodings have equal precision.

`2.93873587705571877e-039` through `1.70141183460469229e+038`

Through this range the `D_FLOAT` encoding has more precision. Eight times as many `D_FLOAT` values fall into this range as do `G_FLOAT` or IEEE 754 values. The `G_FLOAT` and IEEE 754 encodings have equal precision through this range.

`1.70141183460469232e+038` through `8.98846567431157854e+307`

These values can only be represented with the `G_FLOAT` and IEEE 754 encodings. Their magnitudes are too large for the `D_FLOAT` encoding. In this range the `G_FLOAT` and IEEE 754 encodings have equal precision.

`8.98846567431157954e+307` through `1.79769313486231571e+308`

These values can only be represented with the IEEE 754 encoding. Their magnitudes are too large for the `G_FLOAT` and `D_FLOAT` encodings.

`Infinity`

This value exists only in the IEEE 754 encoding.

`NaN`

Not a Number. These non-values exist only in the IEEE 754 encoding.